Driver User Manual

Anybus® CompactCom 40

Doc.Id. HMSI-27-225
Doc. Rev. 1.10

Connecting Devices™

HALMSTAD • CHICAGO • KARLSRUHE • TOKYO • BEIJING • MILANO • MULHOUSE • COVENTRY • PUNE • COPENHAGEN

HMS Industrial Networks
Mailing address: Box 4126, 300 04 Halmstad, Sweden
Visiting address: Stationsgatan 37, Halmstad, Sweden

E-mail: info@hms-networks.com
www.hms-networks.com

Important User Information

This document is intended to provide a good understanding of the driver software that can be used to facilitate development with Active Anybus CompactCom communication modules. Passive modules are not covered, nor does this document cover any network specific features; this information is instead available as separate documents (Network Guides).

The reader of this document is expected to be familiar with high- and low-level software design, and communication systems in general.

The document is optimized for online use; when possible, function calls etc. are implemented as clickable hypertext links.

- The driver does not take over the developer's responsibility to understand how the Anybus CompactCom concept works (i.e. the user is expected to have read and *understood* the Anybus CompactCom 40 Software Design Guide).
- Prior to attempting to use this driver, ensure that the hardware implementation works as expected.
- The user is expected to be familiar with the C programming language.
- The user is expected to be familiar with the development environment used for this project.

For more information, documentation etc., please visit the HMS website, 'www.anybus.com'.

Liability

Every care has been taken in the preparation of this manual. Please inform HMS Industrial Networks AB of any inaccuracies or omissions. The data and illustrations found in this document are not binding. We, HMS Industrial Networks AB, reserve the right to modify our products in line with our policy of continuous product development. The information in this document is subject to change without notice and should not be considered as a commitment by HMS Industrial Networks AB. HMS Industrial Networks AB assumes no responsibility for any errors that may appear in this document.

Those responsible for the use of this software must ensure that all the necessary steps have been taken to verify that the applications meet all performance and safety requirements including any applicable laws, regulations, codes, and standards.

The examples and illustrations in this document are included solely for illustrative purposes. Because of the many variables and requirements associated with any particular implementation, HMS Industrial Networks AB cannot assume responsibility for actual use based on these examples and illustrations.

Intellectual Property Rights

HMS Industrial Networks AB has intellectual property rights relating to technology embodied in the software described in this document. These intellectual property rights may include copyright, patents and pending patent applications in the US and other countries.

Trademark Acknowledgements

Anybus ® is a registered trademark of HMS Industrial Networks AB. All other trademarks are the property of their respective holders.

Source Code Licensing

The Anybus CompactCom 40 Driver source code is subject to the following copyright statement:

© COPYRIGHT NOTIFICATION 2014 HMS Industrial Networks AB

This code is the property of HMS Industrial Networks AB. The source code may not be reproduced, distributed, or used without permission. When used together with a product from HMS, this code can be modified, reproduced and distributed in binary form without any restrictions.

THE CODE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. HMS DOES NOT WARRANT THAT THE FUNCTIONS OF THE CODE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF THE CODE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT DEFECTS IN IT CAN BE CORRECTED.

Table of Contents

Preface **About This Document** **5**
Related Documents 5
Document History 5
Conventions & Terminology 6
Abbreviations 6
Support 6

Chapter 1 **About the Anybus CompactCom 40 Driver** **7**
General Information 7

Chapter 2 **ABCC Driver Stack** **8**

Chapter 3 **Anybus CompactCom Handler Main State Machine** **9**

Chapter 4 **Software Overview** **11**
General 11
Files and Folders 11
Building the Driver 11

Chapter 5 **API** **13**

Chapter 6 **Configuration Parameters** **16**

Chapter 7 **Host Platform Dependent Implementation** **18**

Chapter 8 **Message Handling** **20**
Sending a Command 20
Receiving a Response 21
Receiving a Command 21

Chapter 9 **ADI Mapping** **22**
ADI Entry List 22
Write and Read Process Data Mapping 23
Example Implementations 23

Chapter 10 **Step-by-step User Guide** **24**
General Information 24
Step-by-step 24

Appendix A Message Header Format Conversion 28

Appendix B Using the Driver.....29

Timer System 29

Event Based Applications 29

Application Main Loop 30

Flowcharts 30

P. About This Document

For more information, documentation etc., please visit the HMS website, 'www.anybus.com'.

P.1 Related Documents

Document	Author
Anybus CompactCom M40 Hardware Design Guide	HMS
Anybus CompactCom 40 Software Design Guide	HMS
Anybus CompactCom 40 Drive Profile Design Guide	HMS
Anybus CompactCom 40 Network Guides (separate document for each networking system)	HMS
Anybus CompactCom 40 Drive Profile Network Guides (separate document for each networking system)	HMS

P.2 Document History

Summary of Recent Changes (1.00... 1.10)

Change	Page(s)
Updated support information	6
Revised driver folder structure	11
Added a parameter to the Startdriver command	30

Revision List

Revision	Date	Author	Chapter(s)	Description
0.10	2013-12-19	KeL, KaD	All	New document
0.50	2014-02-06	KaD	All	Revised document
1.00	2014-03-25	KaD	All	Revised document
1.10	2014-06-05	KaD	P, 4, B	Major updates

P.3 Conventions & Terminology

The following conventions are used throughout this document:

- Numbered lists provide sequential steps.
- Bulleted lists provide information, not procedural steps.
- The terms 'Anybus' or 'module' refers to the Anybus CompactCom module.
- The terms 'host', 'host system', or 'host application' is used when referring to the hardware and software that hosts the Anybus CompactCom module.
- Hexadecimal values are written in the format NNNNh or 0xNNNN, where NNNN is the hexadecimal value.
- Intel byte order is assumed unless otherwise stated.
- Host system memory space = memory which can be accessed directly by the host software.
- Parallel interface memory space = memory in which the parallel interface resides; may or may not be available directly in host system memory space.

P.3.1 Abbreviations

Abbreviation	Explanation
ABCC	Anybus CompactCom
ABP	Anybus Protocol
ISR	Interrupt Service Routine
WRPD	Write process data
RDPD	Read process data
WRMSG	Write message
RDMSG	Read message

P.4 Support

For general contact information and support, please refer to the contact and support pages at www.anybus.com.

1. About the Anybus CompactCom 40 Driver

1.1 General Information

The Anybus CompactCom 40 network communication module is a powerful communication solution for demanding industrial applications. It is ideal for high performance, synchronized applications such as servo drive systems. Typical applications are PLCs, HMIs, robotics, AC/DC and servo drives.

The Anybus CompactCom 40 software interface is designed to be network protocol independent, allowing the host application to support all major networking systems using the same software driver, without loss of functionality.

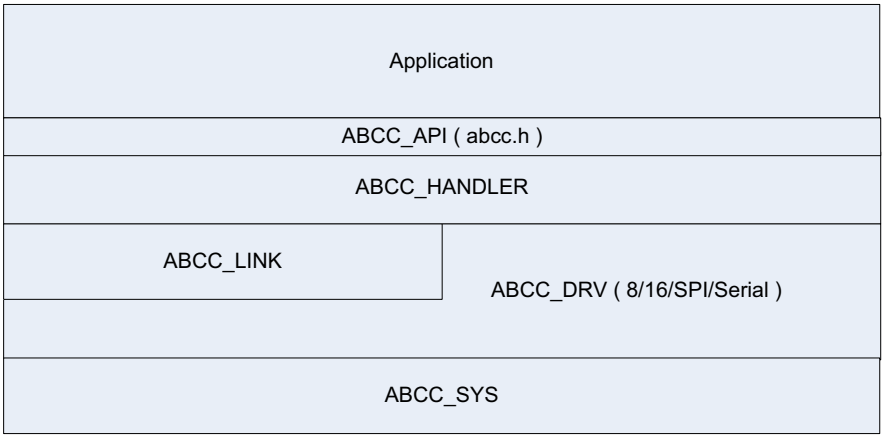
To provide flexibility and room for expansion, an object oriented addressing scheme is used between the host application and the Anybus module. This allows for a very high level of integration, since the host communication protocol enables the Anybus module to retrieve information directly from the host application using explicit object requests rather than memory mapped data.

HMS supplies a free source level (C language) software driver that can be used freely to speed up the development process. The driver acts as "glue" between an Anybus module and the host application, i.e. it separates the low level host interface communication from the host application software and provides easy to use function calls for common Anybus related tasks. It supports one physical interface per application.

The driver is fully OS independent, and can even be used without an operating system if required. Furthermore, it can be used with CompactCom 30 modules as well as CompactCom 40 modules. This manual describes the Anybus CompactCom 40 Driver and gives a step-by-step guide how to implement an application starting from the driver.

In the "Step-by-step User Guide" on page 24, you will find a guide on how to use the driver. The preceding chapters contain all necessary reference documentation.

2. ABCC Driver Stack



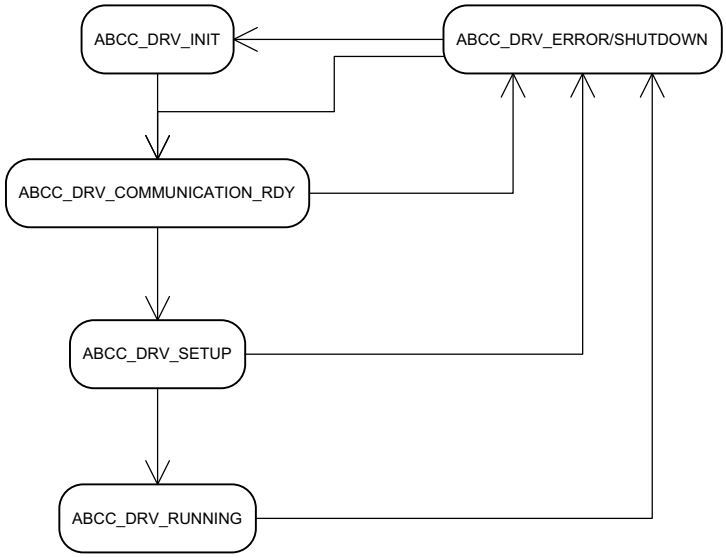
Description of the Driver Parts

- ABCC_API - Provides the common interface for the application using the Anybus CompactCom driver. Interface description can be found in abcc.h.
- ABCC Handler layer - Handling of Anybus CompactCom main state machine driven by events or a polling mechanism. This layer is split up into operation mode dependent and independent parts.
- ABCC_LINK layer - Internal layer implementing message flow control and message buffer handling. Interface description found in abcc_link.h.
- ABCC_DRV - Implements the operation mode specific driver. Interface description found in abcc_drv_if.h.
- ABCC_SYS - Implements the physical interface for a specific operation mode and target. The ABCC_SYS parts contain hardware dependent code that the user needs to implement for the driver to work. The interface descriptions are found in abcc_sys.h, abcc_sys_par.h, abcc_sys_spi.h, abcc_sys_ser.h and abcc_sys_par30.h.

3. Anybus CompactCom Handler Main State Machine

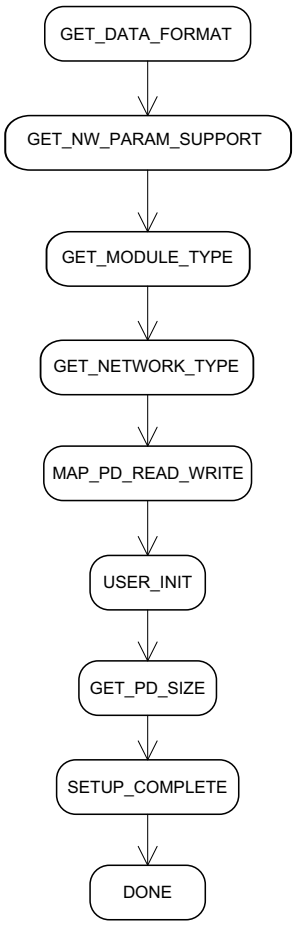
This chapter describes the main states in the Anybus CompactCom handler. The state machine is driven by calling the ABCC_RunDriver() function from the application main loop and/or the ABCC_ISR() function when an event is triggered by the Anybus CompactCom module.

The figure below shows the state machine for the handler main states.



State	Description
ABCC_DRV_INIT	When the function ABCC_StartDriver() is called the following steps will be performed: - All internal parameters and variables are reset. - Module detection is performed - Module identity is read - Operation mode is set - Interrupt is enabled When finished the handler will change states to ABCC_DRV_COMMUNICATION_RDY.
ABCC_DRV_COMMUNICATION_RDY	If power up interrupt indicates that the Anybus module is ready for communication or if the ABCC_USER_STARTUP_TIME_MS timeout has expired, the command ABCC_isReadyForCommunication() will cause a transition to the ABCC_DRV_SETUP state.
ABCC_DRV_SETUP	Implements the Anybus CompactCom setup. The setup is driven by ABCC_RunDriver() or ABCC_ISR(). Once setup is finished, a transition to the RUNNING state is made. The substate machine for the Anybus CompactCom setup is described in “Anybus Setup State Machine” on page 10.
ABCC_DRV_RUNNING	The Anybus CompactCom module is up and running. Incoming events (ABCC_RunDriver() or ABCC_ISR()) are handled or sent to the application.
ABCC_DRV_ERROR	An error has occurred, which has forced the handler to this state. The function ABCC_HwReset() will cause the handler to change states to ABCC_DRV_SHUTDOWN. All other events are ignored.
ABCC_DRV_SHUTDOWN	The function ABCC_StartDriver() will restart the driver.
All states	If an error is detected, the error will be reported and the handler will change states to ABCC_DRV_ERROR. The function ABCC_Shutdown() will force the handler to the state ABCC_DRV_SHUTDOWN.

Anybus Setup State Machine



State	Description
GET_DATA_FORMAT	A message is sent to read the data format. When a correct response is received, a transition to the state GET_NW_PARAM_SUPPORT is performed.
GET_NW_PARAM_SUPPORT	A message is sent to read network parameter support. When a correct response is received, a transition to the state GET_MODULE_TYPE is performed.
GET_MODULE_TYPE	A message is sent to read the module type. When a correct response is received, a transition to the state GET_NETWORK_TYPE is performed.
GET_NETWORK_TYPE	A message is sent to read the network type. When a correct response is received, a transition to the state MAP_PD_READ_WRITE is performed.
MAP_PD_READ_WRITE	The ADI mapping information is retrieved by calling ABCC_CbfAdiMappingReq(). When all read and write mapping has been performed, the state is changed to USER_INIT.
USER_INIT	Any user specific setup can be performed in this state. The setup is triggered by the Anybus CompactCom driver by the call ABCC_CbfUserInitReq(). When ABCC_CbfUserInitComplete() is called by the application, a transition to the state GET_PD_SIZE is performed.
GET_PD_SIZE	In this state the process data size is read from the Anybus Object. When done, a transition to SETUP_COMPLETE is performed.
SETUP_COMPLETE	A message is sent to confirm setup complete. When a correct response is received, a transition to the main state RUNNING is performed.
All states	If any error or corrupted message is detected the appropriate error is reported and a transition to the main state ERROR is made.The error is reported by using ABCC_CbfDriverError().

4. Software Overview

4.1 General

The Anybus CompactCom 40 Driver is a software component that can be completely integrated into a customer product or application to gain access to Anybus CompactCom 40 modules.

Parts of the driver code needs to be adapted, i.e. ported, to the host platform. This generally includes functions which access the Anybus host interface, or functions which needs to be adapted to integrate the driver into the host system. All these functions are found in the files contained in the sys_example folder.

4.1.1 Files and Folders

Folders	Description
$(ROOT)\abcc_app_drv\inc	.h files published to the application. The folder contains driver configuration files for the application as well as for the system dependent part of the driver.
$(ROOT)\abcc_app_drv\src	Anybus CompactCom driver implementation.
$(ROOT)\abcc_app_drv\src\spi	SPI specific files.
$(ROOT)\abcc_app_drv\src\par	8/16 bit event specific files.
$(ROOT)\abcc_app_drv\src\par30	8 bit ping/pong specific files.
$(ROOT)\abcc_app_drv\src\serial	Serial ping/pong specific files.
$(ROOT)\sys_example\sys_template	Implementation examples of the system dependent part of the driver (SDK and empty example), and driver configuration files.
$(ROOT)\abp\inc	ABP header files.

4.1.2 Building the Driver

The following files are required to build the driver.

Published ABCC Driver Interface

File Name	Description
\abcc_app_drv\inc\abcc.h	The public interface for the Anybus CompactCom Driver.
\abcc_app_drv\inc\abcc_ad_if.h	Type definitions for ADI mapping.
\abcc_app_drv\inc\abcc_sys_adapt.h	Interface for target dependent functions common to all operation modes.
\abcc_app_drv\inc\abcc_sys_adapt_spi.h	Interface for target dependent functions needed by abcc_spi_drv.c.
\abcc_app_drv\inc\abcc_sys_adapt_par.h	Interface for target dependent functions needed by abcc_par_drv.c.
\abcc_app_drv\inc\abcc_sys_adapt_ser.h	Interface for target dependent functions needed by abcc_ser_drv.c.

Internal Driver Files

File Name	Description
\abcc_app_drv\src\abcc_drv_if.h	Interface for low level driver implementing the specific operating mode.
\abcc_app_drv\src\abcc_debug_err.h	Help macros for debugging and error reporting.
\abcc_app_drv\src\abcc_link.c \abcc_app_drv\src\abcc_link.h	Message buffer handling and message queue handling.
\abcc_app_drv\src\abcc_mem.c \abcc_app_drv\src\abcc_mem.h	Message resource memory support used by abcc_link.c.

File Name	Description
\abcc_app_drv\src\abcc_handler.c	Anybus CompactCom handler implementation including handler parts that are independent of operation mode.
\abcc_app_drv\src\abcc_setup.c	Anybus CompactCom handler implementation including setup state machine.
\abcc_app_drv\src\abcc_timer.c	Support for Anybus CompactCom driver timeout functionality.

8/16 Bit Parallel Event Specific Files

File Name	Description
\abcc_app_drv\src\par\abcc_handler_par.c	Implements ABCC_RunDriver() and ABCC_ISR().
\abcc_app_drv\src\par\abcc_par_drv.c	Implements abcc_drv_if.h for parallel operating mode.

SPI Specific Files

File Name	Description
\abcc_app_drv\src\par\abcc_handler_spi.c	Implements ABCC_RunDriver() and ABCC_ISR().
\abcc_app_drv\src\spi\abcc_spi_drv.c	Implements abcc_drv_if.h for SPI operating mode.
\abcc_app_drv\src\spi\abcc_crc32.c \abcc_app_drv\src\spi\abcc_crc32.h	Crc32 implementation used by SPI.

8 Bit Parallel Ping/Pong Specific Files

File Name	Description
\abcc_app_drv\src\par\abcc_handler_par30.c	Implements ABCC_RunDriver() and ABCC_ISR().
\abcc_app_drv\src\par\abcc_par30_drv.c	Implements abcc_drv_if.h for parallel 30 ping/pong operating mode.

Serial Specific Files

File Name	Description
\abcc_app_drv\src\par\abcc_handler_ser.c	Implements ABCC_RunDriver() and ABCC_ISR().
\abcc_app_drv\src\serial\abcc_serial_drv.c	Implements abcc_drv_if.h for serial operating mode.
\abcc_app_drv\src\serial\abcc_crc16.c \abcc_app_drv\src\serial\abcc_crc16.h	Crc16 implementation used by Serial.

System Adaptation Interface Files

File Name	Description
\abcc_app_drv\sys_example\sys_template\abcc_td.h	Types used by Anybus CompactCom.
\abcc_app_drv\sys_example\sys_template\abcc_user_def.h	User defines to control and configure the driver.

System Adaptation Implementation Files

File Name	Description
\abcc_app_drv\sys_example\sys_template\abcc_sys_adapt.c	Empty example implementation of system adaptation.
\abcc_app_drv\sys_example\sys_template\abcc_cbf.c	Template for callback functions.
\abcc_app_drv\sys_example\sys_pctransportprovider\ abcc_sys_adapt.c	Implementation of PC transport provider adaptation.

5. API

The Anybus CompactCom API layer defines a common interface for all network applications to the Anybus CompactCom driver. The interface is found in abcc.h.

API Functions

Function	Description
ABCC_StartDriver()	Initiates the driver, enables interrupt, and sets the operation mode. When this function has been called the timer system can be started. Note! This function will NOT release the reset of the module.
ABCC_ShutdownDriver()	Stops the driver and puts it into SHUTDOWN state.
ABCC_IsReadyforCommunication()	This function must be polled after the ABCC_StartDriver() until it returns the value TRUE. This indicates that the module is ready for communication and the ABCC setup sequence is started.
ABCC_RunTimerSystem()	Handles all timers for the ABCC driver. It is recommended to call this function on a regular basis from a timer interrupt. Without this function no timeout and watch-dog functionality will work.
ABCC_RunDriver()	Drives the ABCC driver sending and receiving mechanism. TRUE: Driver is started and ready for communication. FALSE: Driver is stopped or is not started.
ABCC_UserInitComplete()	This function should be called by the application when the last response from the user specific setup has been received. This will end the ABCC setup sequence and ABCC_SETUP_COMPLETE will be sent.
ABCC_SendCmdMsg()	Sends a command message to the module.
ABCC_SendRespMsg()	Sends a response message to the module.
ABCC_SendRemapRespMsg()	Sends a remap response to the module.
ABCC_GetCmdQueueSize()	Retrieves the number of entries left in the command queue.
ABCC_SetAppStatus()	Sets the current application status, according to ABP_AppStatusType in abp.h.
ABCC_MsgAlloc()	Allocates a message buffer. This function MUST be used when allocating mes-sage buffers.
ABCC_MsgFree()	Returns the message buffer to the driver for other messages.
ABCC_HwReset()	Module hardware reset. ABCC_ShutdownDriver() is called from this function. Note! This function will only set reset pin to low. It is the responsibility of the caller to make sure that the reset time (the time between the ABCC_HwReset() and ABCC_HwReleaseReset() calls) is long enough.
ABCC_HwReleaseReset()	Releases the module reset.
ABCC_ReadModuleId()	Reads the module ID. This function can be used before the driver is started and reset is released.
ABCC_ModuleDetect()	Detects if a module is present. This function can be used before the driver is started and reset is released.
ABCC_ModCap()	Reads the module capability. This function is only supported by the ABCC40 par-allel operation mode.
ABCC_LedStatus()	Reads the LED status. Only supported in SPI and ABCC 40 parallel operation mode.
ABCC_AnbState()	Reads the current Anybus state.

API Event Related Functions

Function	Description
ABCC_ISR()	This function should be called from inside the ABCC interrupt routine to acknowledge and handle received ABCC events (triggered by IRQ_n on the ABCC application interface)
ABCC_RdPdEvent()	Triggers a RdPd read.
ABCC_RdMsgEvent()	Triggers a message receive read.
ABCC_NewWrPdEvent()	Indicates that new process data from the application is available and will be sent to the ABCC.

API Callbacks

All these functions need to be implemented by the application.

Function	Description
ABCC_CbfAdiMappingReq()	The function is called when the driver is about to start the automatic process data mapping. It returns mapping information for read and write PD.
ABCC_CbfUserInitReq()	The function is called to trigger a user specific setup during the module setup state.
ABCC_CbfUpdateWriteProcessData()	Updates the current write process data. The data must be copied into the buffer before returning from the function.
ABCC_CbfNewReadPd()	Called when new process data has been received. The process data needs to be copied to the application ADI:s before returning from the function.
ABCC_CbfReceiveMsg()	A message has been received from the module. This is the receive function for all received commands from the module.
ABCC_CbfWdTimeout()	The function is called when communication with the module has been lost.
ABCC_CbfWdTimeoutRecovered()	Indicates a recent ABCC watchdog timeout but now the communication is working again.
ABCC_CbfDriverError()	The driver has detected an error.
ABCC_CbfRemapDone()	This callback is invoked when REMAP response is successfully sent to the module.
ABCC_CbfAnbStatusChanged()	This callback is invoked if the module changes status i.e. if Anybus state or supervision state is changed.
ABCC_CbfEvent()	Called for unhandled events. Unhandled events are events enabled in ABCC_USER_INT_ENABLE_MASK but not present in ABCC_USER_HANDLE_IN_ABCC_ISR_MASK.

Support Functions

Function	Description
ABCC_NetworkType()	Retrieves the network type.
ABCC_ModuleType()	Retrieves the module type.
ABCC_NetFormatType()	Retrieves the network format.
ABCC_DataFormatType()	Retrieves the network endianness.
ABCC_ParameterSupport()	Retrieves the parameter support.
ABCC_GetOpmode()	Calls ABCC_SYS_GetOpmode() to read the operation mode from HW.
ABCC_GetAttribute()	Fills an ABCC message with parameters to get an attribute.
ABCC_SetByteAttribute()	Fills an ABCC message with parameters in order to set an attribute.
ABCC_VerifyMessage()	Verifies an ABCC response message.
ABCC_GetDataTypeSize()	Returns the size of an ABP data type.

6. Configuration Parameters

There are a number of defines in abcc_user_def.h, that are used when configuring the system. They are described in the table below.

Define Name	Description
ABCC_USER_DRV_SPI	SPI operating mode is to be used.
ABCC_USER_DRV_PARALLEL	Parallel event mode is to be used (8/16 bit).
ABCC_USER_DRV_PARALLEL_30	Parallel ping/pong (half duplex) mode is to be used
ABCC_USER_DRV_SERIAL	Serial ping/pong (half duplex) mode is to be used
ABCC_USER_STARTUP_TIME_MS	Time to wait before starting to communicate with the Compact-Com module. Only valid if fast startup is not used. If not defined in abcc_user_def.h, the default value is 1500 ms
ABCC_USER_REMAP_SUPPORT_ENABLED	Define if remap command shall be supported. If defined, ABCC_CbfRemapDone() must be implemented by the application.
ABCC_USER_MAX_NUM_APPL_CMDS	Number of commands that can be sent without receiving any response.
ABCC_USER_MAX_NUM_ABCC_CMDS	Number of commands that can be received without sending any response.
ABCC_USER_MAX_NUM_MSG_RESOURCES	Total number of message resources. If not defined in abcc_user_def.h a default value will be used. The default value is (ABCC_USER_MAX_NUM_APPL_CMDS + ABCC_USER_MAX_NUM_ABCC_CMDS)
ABCC_USER_MAX_MSG_SIZE	Size of the largest message that will be used (in bytes) Note! The CompactCom 30 supports 255 byte messages and the CompactCom 40 supports 1524 byte messages. ABCC_USER_MAX_MSG_SIZE should be set to the largest size that will be sent or received. If this size is not known it is recommended to set this value to the maximum supported size.
ABCC_USER_MAX_PROCESS_DATA_SIZE	Maximum process data size that will be used in either direction (in bytes).
ABCC_USER_MEMORY_MAPPED_ACCESS	The Anybus CompactCom interface is memory mapped. This define will enable the direct memory access macros in sys_adapt_par.h.
ABCC_USER_PARALLEL_BASE_ADR	Defines the base address of the Anybus CompactCom if a memory mapped interface is used.
ABCC_USER_INT_ENABLED	Define if Anybus CompactCom interrupt (IRQ_N pin) shall be used.
ABCC_USER_INT_ENABLE_MASK	Defines what ABCC interrupts shall be enabled (see ABP_INTMASK_X in abp.h) If not defined in abcc_user_def.h, the default mask is 0.
ABCC_USER_HANDLE_IN_ABCC_ISR_MASK	Defines what events should be handled in the interrupt context. If not handled in interrupt, it must be polled using ABCC_RunDriver(). If not defined in abcc_user_def.h the default value will be: ** Parallel 16/8: ABCC_USER_INT_ENABLE_MASK ** Other operation modes N/A
ABCC_USER_POLL_WRPD	If defined, WRPD will be updated (ABCC_CbfUpdateWriteProcessData()) each time ABCC_RunDriver() is called. If not defined, the function ABCC_NewWrPdEvent() must be called by the user to trigger a WrPd update.

Define Name	Description
ABCC_USER_WD_TIMEOUT_MS	Timeout for half duplex (ping pong) watchdog. Time to wait for a correct pong (Par30, Ser). For SPI it means the maximum time to wait for a correct MISO frame. Not applicable for PAR interface.
ABCC_USER_ERR_REPORTING_ENABLED	Enable the error reporting callback function. (ABCC_CbfDriverError())
ABCC_USER_DEBUG_PRINT	Macro is used by driver for debug prints such as events or error debug information. If not defined the driver will be silent.
ABCC_USER_64BIT_ADI_SUPPORT	64 bit ADI is supported.
ABCC_USER_DEBUG_EVENT_ENABLED	Debug printouts are enabled.
ABCC_USER_DEBUG_ERR_ENABLED	Debug information printouts are enabled when ABCC_CbfDriverError() is called.
ABCC_USER_BIG_ENDIAN	Big endian host is used. If not defined, little endian is assumed.

Define (SPI specific)	Description
ABCC_USER_SPI_MSG_FRAG_LEN	Length of SPI message fragment in each transaction (in bytes) If shorter than the largest message the sending or receiving of a message may take several SPI transactions. Each SPI transaction will have a message field of this length regardless if a message is present or not. If messages are important then the fragment length should be set to the largest message to avoid fragmentation. If IO data is important the message fragment length could be set to a smaller value.
ABCC_USER_SPI_MAX_PD_LEN	Max number of 16 bit words allocated for the process field in the SPI frame. **Note**: Only the actually mapped process data length will be transmitted in the SPI frame. Since the MOSI and MISO SPI frames need to be equal in length, the maximum of WrPd and RdPd mapped size will be chosen.

7. Host Platform Dependent Implementation

All platform dependent functions are gathered in the abcc_sys_adapt* files. These functions are used solely by the driver, and not by the application.

Anybus CompactCom System Dependent Functions

Function	Description
ABCC_SYS_Init()	This function is called by the driver at the beginning of ABCC_StartDriver(). If needed, any hardware or system dependent initialization shall be done here. Note that ABCC_StartDriver() will also be called during restart of the driver.
ABCC_SYS_Close()	Called from driver at the end of ABCC_ShutDown(). Note that the driver can be started again by calling ABCC_StartDriver().
ABCC_SYS_SetOpmode()	Sets ABCC Operating Mode pins on the ABCC interface. In case operating mode is fixed or set by dip switches this function could be left empty or used to verify the expected operating mode.
ABCC_SYS_GetOpmode()	Read current ABCC Operating Mode from HW.
ABCC_SYS_HWReset()	Reset the module. Set the reset pin on the ABCC interface to low.
ABCC_SYS_HWReleaseReset()	Release reset of the module. Sets the reset pin on the ABCC interface to high.
ABCC_SYS_ReadModuleId()	Read Module Identification pins on the host connector. If the identification pins are not connected this function must return the correct value corresponding to the device. 0 0 (0) Active CompactCom 30 series 0 1 (1) Passive CompactCom 1 0 (2) Active CompactCom 40 series 1 1 (3) Customer specific
ABCC_SYS_ModuleDetect()	Detects if a module is present by reading the Module Detection pins on the ABCC interface. If the detection pins are not connected this function must return true.
ABCC_SYS_AbccInterruptEnable()	Enable the ABCC HW interrupt (IRQ_N pin on the application interface). This function will be called by the driver when the ABCC interrupt shall be enabled.
ABCC_SYS_AbccInterruptDisable()	Disable ABCC HW interrupt (IRQ_N pin on the application interface)
ABCC_SYS_EnterCritical()	This function is called by the driver when there is a possibility of internal resource conflicts between the ABCC interrupt handler and the application thread or main loop. The function temporary disables interrupts to avoid conflict. Note that all interrupts that could lead to a driver access need to be disabled. If no interrupts are used and the driver is accessed by a single thread or main loop there is no need to implement this function. This function could also protect from conflicts caused by different threads accessing the driver at the same time by disabling all interrupts.
ABCC_SYS_ExitCritical()	Restore interrupts to the state when ABCC_SYS_EnterCritical() was called.
ABCC_SYS_UseCritical()	If any preparation is needed before calling ABCC_SYS_EnterCritical() or ABCC_SYS_ExitCritical(), this macro is used to add HW specific necessities.

Parallel (8/16) Specific Functions

Function	Description
ABCC_SYS_ParallelRead()	Reads an amount of bytes from the ABCC memory.
ABCC_SYS_ParallelRead8()	Reads a byte from the ABCC memory. In case of a memory mapped system this function does not need not be implemented.
ABCC_SYS_ParallelRead16()	Reads a word from the ABCC memory. In case of a memory mapped system this function does not need not be implemented.
ABCC_SYS_READ8() ABCC_SYS_READ16()	The driver will use the ABCC_SYS_READ8 and ABCC_SYS_READ16 macros to access the ABCC registers.
ABCC_SYS_ParallelWrite()	Writes an amount of bytes to the ABCC memory.
ABCC_SYS_ParallelWrite8()	Writes a byte to the ABCC memory. In case of a memory mapped system this function does not need not be implemented.
ABCC_SYS_ParallelWrite16()	Writes a word to the ABCC memory. In case of a memory mapped system this function does not need not be implemented.
ABCC_SYS_CopyMemBytes()	Copy a number of bytes, from the source pointer to the destination pointer.
ABCC_SYS_WRITE8() ABCC_SYS_WRITE16()	The driver will use the ABCC_SYS_WRITE8 and ABCC_SYS_WRITE16 macros to access the ABCC registers.
ABCC_SYS_ParallelGetRdPdBuffer()	Get the address to the received read process data.
ABCC_SYS_ParallelGetWrPdBuffer()	Get the address to store the write process data.

SPI Specific Functions

Function	Description
ABCC_SYS_SpiRegDataReceived()	Used in SPI mode. Registers the callback function that will be called when new data is received.
ABCC_SYS_SpiSendReceive()	Used in SPI mode. Handles sending and receiving data in SPI mode.

Serial Specific Functions

Function	Description
ABCC_SYS_SerRegDataReceived()	Used in serial mode. Registers a callback function that indicates that new data has been received on the serial channel.
ABCC_SYS_SerSendReceive()	Used in serial mode. Send TX telegram and prepare for RX telegram reception.
ABCC_SYS_SerRestart()	Used in serial mode. Restart the serial driver. Typically used when telegram has timed out.

8. Message Handling

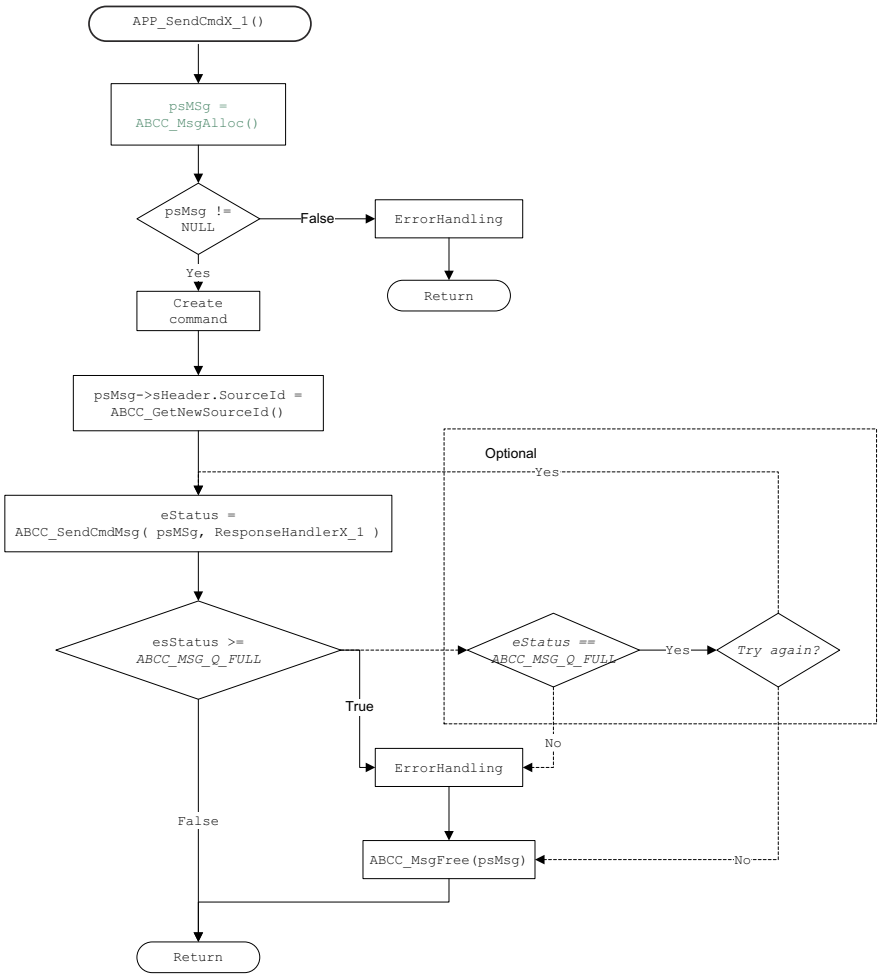
The driver provides a message sending and receiving mechanism. When a message can't be sent to the CompactCom module directly it will be put in a queue and sent as soon the module is ready to receive it. There are two message queues, one for response messages and one for command messages. The response queue has higher priority and will be checked first. Each queue is handled in a FIFO manner.

8.1 Sending a Command

If a command message shall be sent to the module the message buffer must be allocated using the ABCC_MsgAlloc() function provided by the driver API (abcc.h). See ABCC_SendCmdMsg() in "API Functions" on page 13 for more information. The flowchart below shows an example of this.

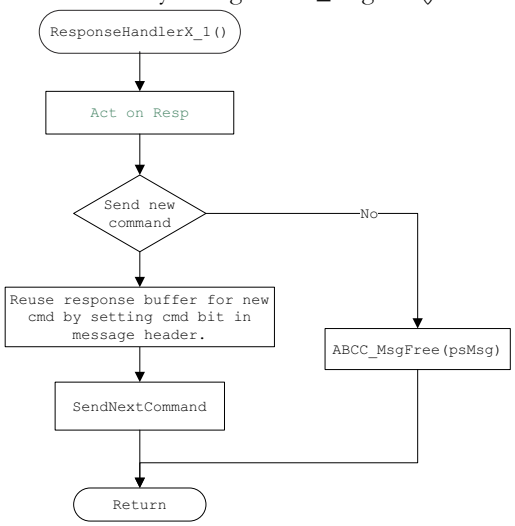
Implement a function like APP_SendCmdX_1(). In this example the source ID is created using the ABCC_GetNewSourceId() function. This ensures that a unique source ID is used. When the response is received the source ID is associated with the ResponseHandlerX_1() function that will be called by the driver.

If there are more message resources than the driver can handle simultaneously there is a possibility that the sending queue is full. This is described in the optional part in the flowchart. In any case, if the driver has not accepted the message the buffer needs to be freed by the user or resent later.



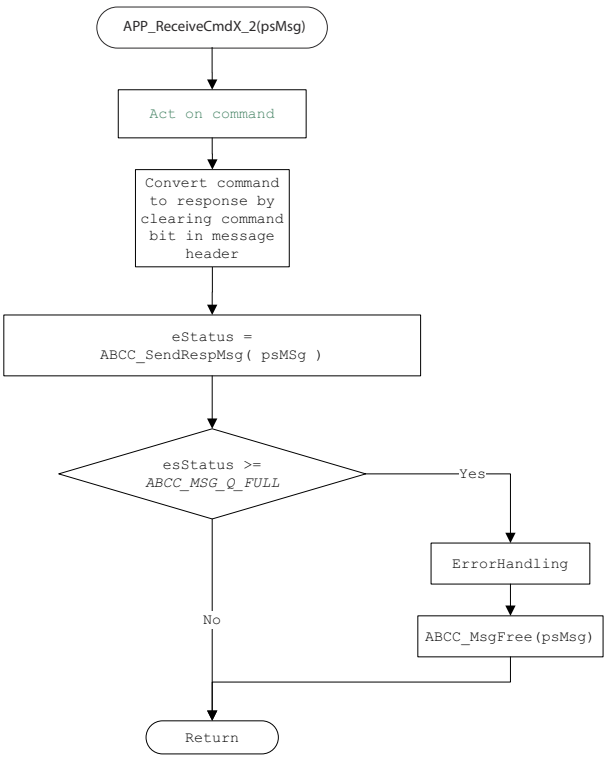
8.2 Receiving a Response

Every time a command is sent, a function is provided to handle the response (a response handler). The flowchart below shows a simple example of how to handle a response. If the response triggers a new command, the message buffer can be reused. If there is no new command, the receive buffer memory must be freed by calling ABCC_MsgFree().



8.3 Receiving a Command

The flowchart below shows an example of when a command is received. The buffer for the received command can be reused for the response. If the driver does not accept the response the caller needs to free the message buffer to avoid memory leaks.



9. ADI Mapping

Type definitions for creating your ADI mapping are found in abcc_ad_if.h.

9.1 ADI Entry List

An example of what an ADI mapping table can look like is shown below:

```
const AD_AdiEntryType AD_asADIEntryList[] =
{
/* Idx:0 */ { 1, "RadOnOff",            ABP_BOOL,  1, ALL_ACCESS, &VAPP_fRadOnOff,        &VAPP_BOOL8Props_fRadOnOff },
/* Idx:1 */ { 2, "RadSetTemp",          ABP_FLOAT, 1, ALL_ACCESS, &VAPP_rRadSetTemp,      &VAPP_FLOAT32Props_fRadSetTemp },
/* Idx:2 */ { 3, "RadErrCode",          ABP_ENUM,  1, ALL_ACCESS, &VAPP_eRadErrCode,      &VAPP_ENUMProps_eRadErrCode },
/* Idx:3 */ { 4, "FanOnOff",            ABP_BOOL,  1, ALL_ACCESS, &VAPP_fFanOnOff,        &VAPP_BOOL8Props_fFanOnOff },
/* Idx:4 */ { 5, "FanOpErrCode",        ABP_ENUM,  1, ALL_ACCESS, &VAPP_eFanErrCode,      &VAPP_ENUMProps_eFanErrCode },
/* Idx:5 */ { 6, "ActTemp",             ABP_FLOAT, 1, ALL_ACCESS, &VAPP_rActTemp,         &VAPP_FLOAT32Props_rActTemp },
/* Idx:6 */ { 500, "ActTempErrCode",    ABP_ENUM,  1, ALL_ACCESS, &VAPP_eActTempErrCode,  &VAPP_ENUMProps_ActTempErrCode },
/* Idx:7 */ { 501, "InputTemp",         ABP_FLOAT, 1, ALL_ACCESS, &VAPP_rInputTemp,       NULL },
/* Idx:8 */ { 502, "HeatEffect_deg/sec", ABP_FLOAT, 1, ALL_ACCESS, &VAPP_rHeatEff,       NULL },
/* Idx:9 */ { 503, "SystemTripBits",    ABP_UINT8, 1, ALL_ACCESS, &VAPP_bTripBits,        NULL },
};
```

Each entry is structured as follows:

```
/* Idx:x */ { iInstance, pabName, bDataType, bNumOfElements, bDesc, pxValuePtr, pxValueProps }
```

The entry items are described in the table below.

ADI Entry Item	Description
iInstance	ADI instance number (1-65535). 0 is reserved for Class.
pabName	Name of ADI (character string, ADI instance attribute #1). If NULL, a zero length name will be returned.
bDataType	ABP_BOOL: Boolean ABP_SINT8: 8-bit signed integer ABP_SINT16: 16-bit signed integer ABP_SINT32: 32-bit signed integer ABP_UINT8: 8-bit unsigned integer ABP_UNIT16: 16-bit unsigned integer ABP_UINT32: 32-bit unsigned integer ABP_CHAR: Character ABP_ENUM: Enumeration ABP_SINT64: 64-bit signed integer ABP_UINT64: 64-bit unsigned integer ABP_FLOAT: Floating point value (32 bits)
bNumOfElements	Number of elements of the data type specified in bDataType.
bDesc	Entry descriptor. Bit values according to the following configurations: ABP_APPD_DESCR_GET_ACCESS: Get service is allowed on value attribute. ABP_APPD_DESCR_SET_ACCESS: Set service is allowed on value attribute. ABP_APPD_DESCR_MAPPABLE_WRITE_PD: Remap service is allowed on value attribute. ABP_APPD_DESCR_MAPPABLE_READ_PD: Remap service is allowed on value attribute. The descriptors can be logically OR:ed together. In the example, ALL_ACCESS is all of the above logically OR:ed together.
pxValuePtr	Pointer to local value variable. The type is dependent on bDataType.
pxValueProps	Pointer to local value properties struct, if NULL, no properties are applied (max/min/default). The type is dependent on bDataType.

9.2 Write and Read Process Data Mapping

ADIs to be mapped are defined with an 0xFFFF terminated list. There is one combined list for read process data and write process data. Each element in the list contains the ADI index (Note: not the instance number) for the ADI in AD_asADIEntryList, and if the mapping is in the read or write direction. Entries in each list are the indices for the ADI in AD_asADIEntryList.

```
static const AD_DefaultMapType AD_asDefaultMap[] =
                   {{0, PD WRITE},
                    {3, PD READ}
                    {1, PD WRITE},
                    {4, PD READ},
                    {2, PD WRITE},
                    {5, PD READ},
                    {0xFFFF}};
```

During the setup sequence the Anybus CompactCom driver will ask for this information by invoking ABCC_CbfAdiMappingReq().

9.3 Example Implementations

For mapping the example above the implementation looks like this:

```
void ABCC_CbfAdiMappingReq(   const AD_AdiEntryType** const ppsAdiEntry,
                              const AD_DefaultMapType**  const ppsDefaultMap )
{
   *ppsAdiEntry = AD_asADIEntryList;
   *ppsDefaultMap = AD_asDefaultMap;
}
```

If no mapping should be done during the setup phase the implementation looks like:

```
void ABCC_CbfAdiMappingReq(   const AD_AdiEntryType** const ppsAdiEntry,
                              const AD_DefaultMapType**  const ppsDefaultMap )
{
   *ppsAdiEntry = NULL;
   *ppsDefaultMap = NULL;
}
```

10. Step-by-step User Guide

This chapter contains a step-by-step guide on how to set up a simple example application using the Anybus CompactCom 40 Driver, with relatively little effort. It does not provide a finished implementation, but should be considered as a starting point to better understand the procedures involved when using the driver and when implementing an application of your own.

10.1 General Information

In many applications, the standard driver can be used without many changes. Performing the following steps will result in a simple working application. Each step is described in further detail in the following sections of this chapter.

Before you begin, some advice:

- The C compiler should be C90 compatible.
- Study the Anybus CompactCom 40 Software Design Guide to get an understanding of the Anybus CompactCom concept.
- Ensure that the hardware implementation works as expected.
- You should be familiar with the C programming language.
- You should be familiar with the development environment for the product.
- When programming, make good use of the reference part of this manual and comments included in the source code, mainly abcc.h, abcc_sys_adapt.h and abcc_ad_if.h.

10.2 Step-by-step

The steps to follow are:

1. Preparations to set up the project.
2. Configure the driver.
3. Implement system specific files or choose a previously existing implementation.
4. Remove unused code.
5. Write application.
   - Map default ADIs (if needed).
   - Implement user specific setup (not mandatory).
6. Compile and run application.

Step 1: Preparations

Before modifying any driver files, the project has to be set up properly.

- **Create an empty project**
- **Copy the driver files into the project folder**
- **Add the driver files to the project**

- **Rename abcc_user_def_example.h to abcc_user_def.h**

  If you update an older version of the driver it is recommended to differ the old abcc_user_def.h and the new abcc_user_def.h.

- **Create a new file and add a simple main function**

```
#include "abcc_td.h"
#include "abcc.h"
#include "abp.h"
#include "abcc_ad_if.h"

main()
 {
   //nothing yet
 }
```

- **Modify the file 'abcc_td.h'.**

  This file contains various types used by the driver. Please ensure that the standard 8, 16 and 32 bit datatypes, both in signed and unsigned formants, are correct for your compiler.

- **Compile**

  Verify that the project compiled without errors.

Step 2: Configure the Driver

The user #defines described in abcc_user_def.h are used to setup and configure the system, defining constants such as timeouts, buffer sizes, queue sizes, polling methods and interrupt methods. The file abcc_user_def.h is included where necessary in the driver files.

Basic Defines to Include in the Application

- **Set time for startup**

  The #define ABCC_USER_STARTUP_TIME_MS specifies the time from reset release to module ready for communication, and should be set to 1500 ms. Only valid if powerup interrupt is not used to indicate that the module is ready for communication.

- **Set interrupt communication**

  The #define ABCC_USER_INT_ENABLE specifies if the Anybus CompactCom interrupt (IRQ_N pin) shall be used. Must not be used in serial communication.

- **Set buffer sizes and queue sizes**

  The following defines will decide the sizes of buffers and queues.
  - #define ABCC_USER_MAX_NUM_APPL_CMDS
  - #define ABCC_USER_MAX_NUM_ABCC_CMDS
  - #define ABCC_USER_MAX_MSG_SIZE
  - #define ABCC_USER_MAX_PROCESS_DATA_SIZE

- **Decide polling method**
  - #define ABCC_USER_POLL_WRPD
    If defined, WRPD will be updated by ABCC_CbfUpdateWriteProcessData() each time ABCC_RunDriver() is called. If not defined, ABCC_NewWrPdEvent() must be called by the user to trigger a WRPD update.

    It is recommended to define ABCC_USER_POLL_WRPD and use the ABCC_RunDriver() function to request new WRPD data.

Defines for Specific Operation Modes

- **Parallel mode (8/16 bit, event driven)**
  - Define #define ABCC_USER_DRV_PARALLEL
  - Set #define ABCC_USER_INT_ENABLE_MASK to define which interrupts are enabled

  See "Configuration Parameters" on page 16 and "Event Based Applications" on page 29 for more information.

- **SPI mode (event driven)**
  - Define #define ABCC_USER_DRV_SPI
  - Define #define ABCC_USER_ABCC_INT_ENABLE

  Set the following #defines to setup SPI communication:
  - #define ABCC_USER_SPI_MSG_FRAG_LEN
  - #define ABCC_USER_WD_TIMEOUT_MS

  See "Define (SPI specific)" on page 17 for more information.

- **Serial mode (UART, half duplex)**
  - Define #define ABCC_USER_DRV_SERIAL
  - Set #define ABCC_USER_WD_TIMEOUT_MS to an appropriate timeout value. See "ABCC_USER_WD_TIMEOUT_MS" on page 17 for more information.

- **Parallel 30 mode (8 bit, half duplex)**
  - Define #define ABCC_USER_DRV_PARALLEL_30
  - Define #define ABCC_USER_INT_ENABLE if needed
  - Set #define ABCC_USER_WD_TIMEOUT_MS to an appropriate timeout value. See "ABCC_USER_WD_TIMEOUT_MS" on page 17 for more information.

Step 3: Implement system specific settings

- **Modify the host platform adaptation files**

  See "System Adaptation Interface Files" on page 12 and "Host Platform Dependent Implementation" on page 18. The functions in 'abcc_sys_adapt.h' and in 'abcc_sys_adapt_par.h', 'abcc_sys_adapt_spi.h', 'abcc_sys_adapt_par30.h' or 'abcc_sys_adapt_ser.h' defines the interface to the hardware specific parts for the used target.

  If you do not change any of these files, you will have an example implementation of a system adaptation. An empty implementation is available in sys_template/abcc_sys_adapt.c.

  If there is a finished implementation available, you may select this.

- **Compile**

  Verify that the project compiled without errors.

Step 4: Remove unused code

The driver contains much code, that does not have to be used in the specific application. For example, unused operation modes and their corresponding adaption files (see "Files and Folders" on page 11). Remove this code from the project. Verify that the project compiles without errors.

Step 5: Write application

Map default ADIs

If you need to map your own ADIs, do so now. See "ADI Mapping" on page 22.

Implement user specific setup

Implement the user specific parts in the ANB_SETUP phase, like for instance setting the Profibus slave address. Data that can be read from the network should be initialised at this point at the latest. After this state of the Anybus Setup State Machine an end-user could read data at any time.

(User specific implementation is not mandatory, but note that ABCC_UserInitComplete() must be called in any case.) See "Anybus Setup State Machine" on page 10 for more information.

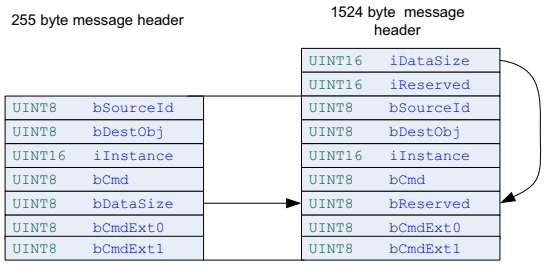
All necessary setup of #defines has now been made and you can write an application.

Step 6: Compile and Run

A. Message Header Format Conversion

The message header format conversion is done inside the driver and the user does not need to take any action. The conversion is simple and only involves copying the data size field to the correct position in the header.

One improvement of the 40 series module is the ability to handle up to 1524 bytes of messaging data. This improvement required the data size indicator of the message header to be increased to 16 bits. The message header supporting 1524 byte messages differs from the old format since the size field need to be 16 bits. The driver supports communication with the 30 module as well as the 40 module, but only supports the new message format in the driver API. If an Anybus CompactCom 30 module is used the driver will internally convert to the legacy message format. The figure below shows the two message formats.



B. Using the Driver

B.1 Timer System

It is recommended to set up a timer interrupt that calls ABCC_RunTimerSystem() on a regular basis.

```
interrupt MyTenMSTimerInterrupt( void )
{
   ABCC_RunTimerSystem( TimeInMsSinceLastCall );
}
```

Or, alternatively, make it part of a main polling loop.

```
while( True )
{
   Delay(10)
   ABCC_RunTimerSystem( 10 );
..
}
```

B.2 Event Based Applications

To be able to run an event based application the operating mode must be parallel 8/16 bit.

For event based applications the ABCC_ISR() must be called from the interrupt routine serving the ABCC interrupt.

The following defines in user_def.h must be defined.

```
#define ABCC_USER_INT_ENABLED
#define ABCC_USER_INT_ENABLE_MASK ( ABP_INTMASK_RDPDIEN | ABP_INTMASK_RDMSGIEN
                                    ABP_INTMASK_WRMSGIEN | ABP_INTMASK_ANBRIEN
                                   ABP_INTMASK_ANBRIEN | ABP_INTMASK_STATUSIEN
                                   )
```

The ABCC_USER_INT_ENABLED enables interrupt handling in the driver.

The ABCC_USER_INT_ENABLE_MASK controls the actual interrupts invoking the ABCC_ISR(). The interrupt mask defines could be found in abp.h.

By default, all interrupts defined in ABCC_USER_INT_ENABLE_MASK are handled directly in ABCC_ISR(). If the user wants to handle only some of the interrupts, it is possible to define the events that should be handled by ABCC_ISR().

```
#define ABCC_USER_HANDLE_IN_ISR_MASK        ABP_INTMASK_RDPDIEN
```

In the example above the read process data will be handled in the interrupt. All other events will be passed to the application by invoking ABCC_CbfEvent().

The driver supports both edge and level triggered interrupts. (The ABCC_ISR() function will execute until the interrupt status register is cleared for all enabled events).

B.3 Application Main Loop

This section gives an example of a simple main loop.

```
void main( void )
{

/* The operation mode is set */
   ABCC_StartDriver( opMode, fPingPong );

/* Release reset of the Anybus CompactCom module */
   ABCC_HwReleaseReset();

/* Enable timer interrupt */

   while( !isReadyForCommunication() );

/* The SETUP sequence has started and interrupts are enabled if configured */
```

If the driver is polled start the polling here.

```
/*
** Start running the driver
*/
   while(ABCC_RunDriver();
   {

      Delay( 1ms );

   }

/*
** Stop the driver
*/

/* Disable the 10 ms interrupt */
   ABCC_ShutdownDriver();

/*Reset of the Anybus CompactCom module */
   ABCC_HwReset();
```

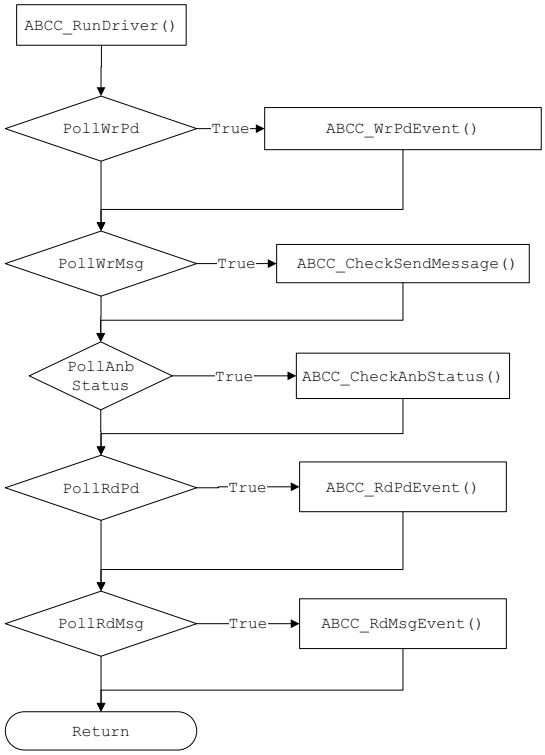
For a completely event based driver, the ABCC_RunDriver() call can be skipped since ABCC_ISR() will take care of all tasks and events.

B.4 Flowcharts

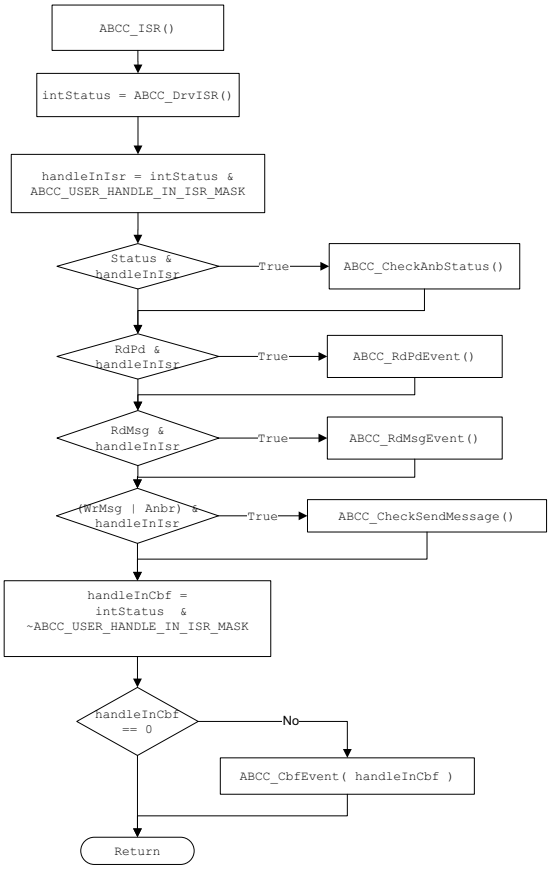
Below are flowcharts for the main functions:

- Event Driven Parallel RunDriver Flow
- Event Driven Parallel Interrupt Service Routine
- Parallel 30 and Serial (Without Interrupt) RunDriver Flow
- Parallel 30 (With Interrupt) RunDriver Flow
- Parallel 30 (With Interrupt) Interrupt Service Routine
- SPI RunDriver Flow
- SPI Interrupt Service Routine
- Read Message Data Flow
- Read Process Data Flow
- Write Process Data Flow

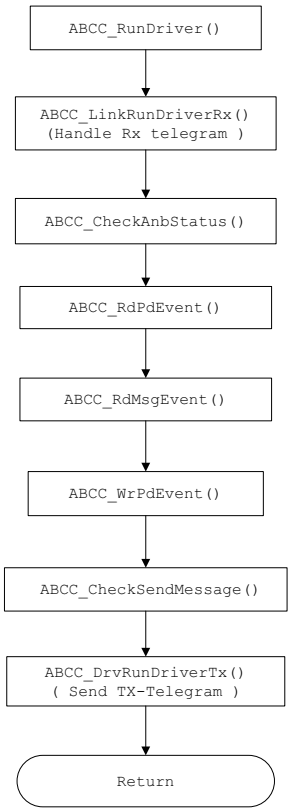
Event Driven Parallel RunDriver Flow



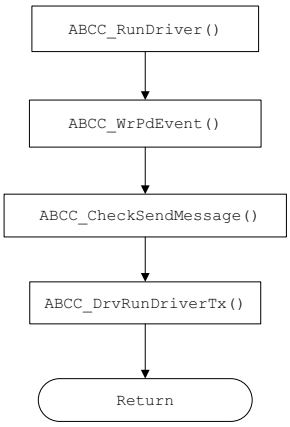
Event Driven Parallel Interrupt Service Routine



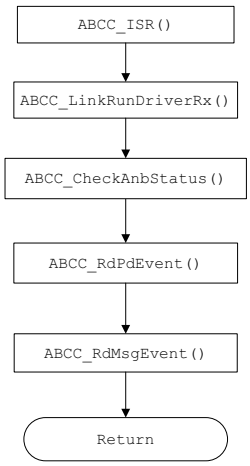
Parallel 30 and Serial (Without Interrupt) RunDriver Flow



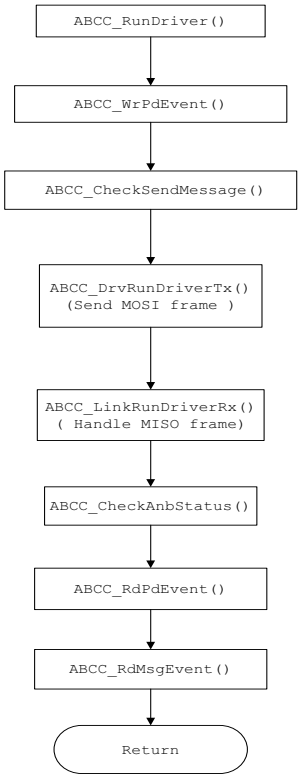
Parallel 30 (With Interrupt) RunDriver Flow



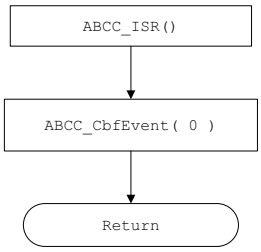
Parallel 30 (With Interrupt) Interrupt Service Routine



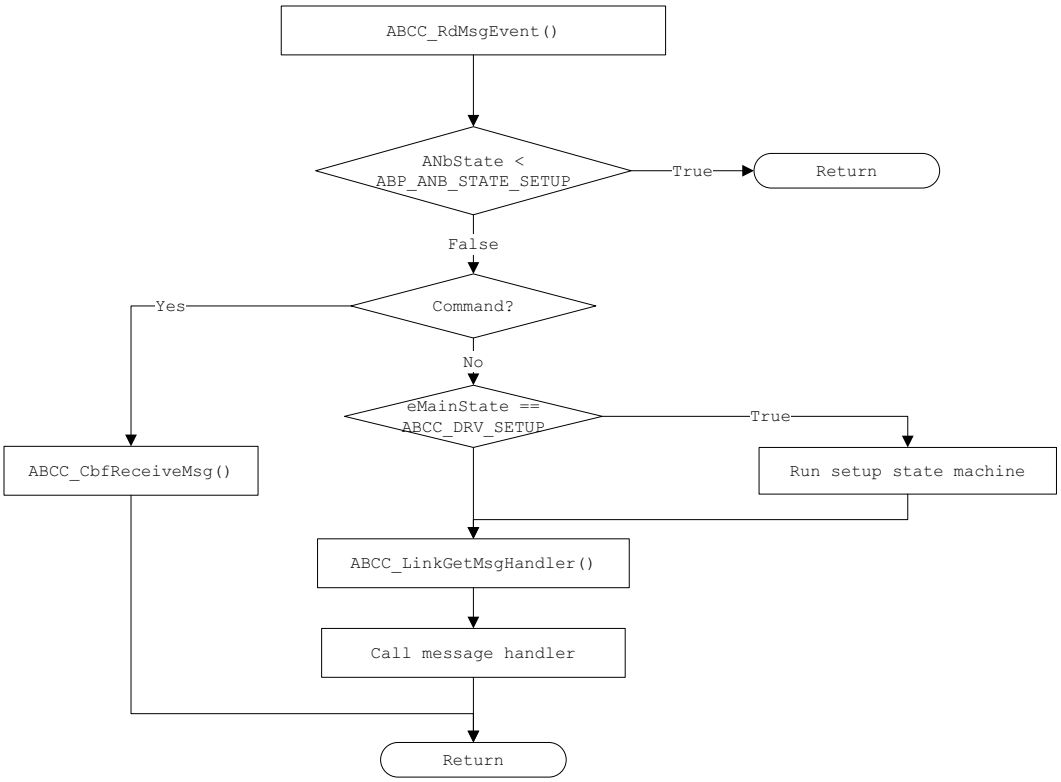
SPI RunDriver Flow



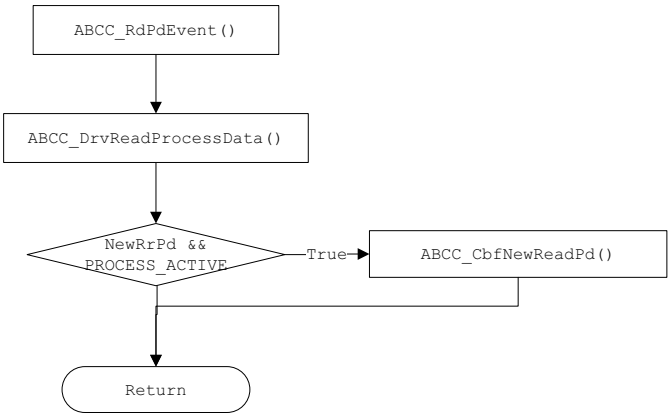
SPI Interrupt Service Routine



Read Message Data Flow



Read Process Data Flow



Write Process Data Flow